

Data Assimilation using a GPU Accelerated Path Integral Monte Carlo Approach

John C. Quinn

*Department of Physics
and
BioCircuits Institute*

Henry D.I. Abarbanel

*Department of Physics,
Marine Physical Laboratory (Scripps Institution of Oceanography)
and
Center for Theoretical Biological Physics
University of California, San Diego
La Jolla, CA 92093-0402 USA*

Abstract

The answers to data assimilation questions can be expressed as path integrals over all possible state and parameter histories. We show how these path integrals can be evaluated numerically using a Markov Chain Monte Carlo method designed to run in parallel on a Graphics Processing Unit (GPU). We demonstrate the application of the method to an example with a transmembrane voltage time series of a simulated neuron as an input, and using a Hodgkin-Huxley neuron model. By taking advantage of GPU computing, we gain a parallel speedup factor of up to about 300, compared to an equivalent serial computation on a CPU, with performance increasing as the length of the observation time used for data assimilation increases.

Keywords: Data assimilation, State and parameter estimation, GPU computing, Path integral Monte Carlo, Hodgkin-Huxley

1. Introduction

Data assimilation is designed to utilize information from measurements of an observed physical or biological system to complete a model of that system. Completion means accurately estimating unknown parameters in the model and accurately estimating the unobserved model state variables at the end, time T , of the observation window. The information on the parameters \mathbf{p} and the full

Email addresses: jquinn@ucsd.edu (John C. Quinn), habarbanel@ucsd.edu (Henry D.I. Abarbanel)

set of model state variables \mathbf{x}_T allow forecasting the state of the system using the model. Also knowledge of the state \mathbf{x}_T permits assessing the quality of the model.

Typically the model will take the form of a system of nonlinear differential equations, either based on the underlying physics of the system in question, or designed to reproduce observed phenomena. To make quantitative predictions the parameters of the model need to be adjusted to fit the data. Since neither the model nor the measurements are exact, the problem should be treated probabilistically. The knowledge we gain from the observations allows us to estimate the conditional probability distribution $P(\mathbf{x}_T|\mathbf{y}_{1:M})$, where $\mathbf{y}_{1:M} = \{\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_M\}$ is the collection of observations up to time $t_M = T$. Moments and marginal distributions of parameters or state variables may also be estimated.

Within an observation window, one makes measurements at time $t_n = \{t_1, t_2, \dots, t_M = T\}$. At each time t_n , L observations are made, represented by the L -dimensional vector $\mathbf{y}(t_n) = \mathbf{y}_n$. The observations provide information about the system state, but it is not complete information, because the observations are likely to be noisy, and more importantly, the dimension L of \mathbf{y} is typically less than the dimension of the system state $x_{n,a}$; $a = 1, 2, \dots, D \geq L$.

The problem of data assimilation was formulated in a probabilistic way some time ago [1, 2]. The key question is this: given a time series of observations $\mathbf{Y} = \mathbf{y}_{1:M} = \{\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_M\}$ and a model, what model state histories $\mathbf{X} = \{\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_M\}$ and set of parameters $\mathbf{p} = \{p_1, p_2, \dots, p_{N_p}\}$ could have produced the observed data? The conditional distribution over state histories and conditioned on observations is called $P(\mathbf{X}|\mathbf{Y})$.

Various methods have been developed to solve this problem. Kalman filtering [3] is the exact solution when the model is linear and the noise is Gaussian and white in time. A more general and computationally intensive approach is particle filtering [4, 5, 6], where the distribution over states is approximated by an ensemble of particles. A similar approach is to phrase the problem in terms of continuous time path integrals [7, 8, 9]. The continuous time path integrals are approximations, and in discrete time an exact formulation has been given [10]. In the path integral approach, the whole time history of states \mathbf{X} is considered at once, instead of sequentially calculating the states at discrete time steps, \mathbf{x}_n .

What we really want are moments of functions on the path \mathbf{X} or histograms of the marginal distributions, $P(x_{n,l}|\mathbf{Y})$, where $x_{n,l}$ is one component of \mathbf{X} . These can be expressed as path integrals over all possible state histories \mathbf{X} , with each path weighted by $P(\mathbf{X}|\mathbf{Y})$. The path integrals can then be approximated using standard Markov Chain Monte Carlo techniques. The main difficulty with this approach is that the methods typically require very many iterations to get good results, and so they can be very computationally expensive.

This problem can be ameliorated by taking advantage of parallel computing technology using a graphics processing unit (GPU). These computations can be done on inexpensive desktop computers, with an off-the-shelf GPU, which can execute hundreds of threads in parallel (see [11] for another use of GPU computing in physics). The Markov chain still must be updated sequentially of

course, but the computations done on each iteration can be highly parallelized. Since in the path integral approach, all time steps are considered at once, it is possible to parallelize the algorithm such that each thread does the computations associated with a different time step. This leads to a large decrease in the amount of computation time required, and makes many problems become practical. We show an example problem using simulated neuron voltage time series data, and the Hodgkin-Huxley neuronal model as inputs into the procedure. In this example we get a parallel speedup factor of up to about 300.

2. Path Integral Formulation of Data Assimilation

This section is review of previous work [1, 2, 10, 12, 13] and an introduction of notation. The idea is to express the answer to probabilistic data assimilation questions as path integrals over all possible state histories, and all possible parameter values.

The inputs into this formulation will be a model of the observed system and a time series of observations, together with uncertainties associated with these two ingredients. The model will be in the form of Markov transition probabilities, which describe how the model state evolves in time. The transition probabilities may come from a discrete time map or a system of ordinary differential equations which is then discretized. In this section we will show how these ingredients are combined to form a probability distribution that is a function of the time history of the model state conditioned on the observations.

The first ingredient is the model. To make a model, we first represent the state of the system at each time step $n \in \{0, 1, \dots, M\}$ as a D -dimensional vector \mathbf{x}_n . We assume the dynamical model is Markov, and so we can represent the time evolution using a transition probability $P(\mathbf{x}_{n+1}|\mathbf{x}_n)$ which depends only on \mathbf{x}_{n+1} and \mathbf{x}_n , and not any previous states. This tells us the probability of transitioning from state \mathbf{x}_n at time step n to state \mathbf{x}_{n+1} at time step $n+1$. If there is no noise in the model, this is a deterministic transition in which case $P(\mathbf{x}_{n+1}|\mathbf{x}_n)$ will be a delta function. When noise is present, typically due to a noisy environment or arising from resolution errors associated with discretization of the model in time and space, this will be represented as stochastic transition in which case $P(\mathbf{x}_{n+1}|\mathbf{x}_n)$ will be a broader version of a delta function, with a width which depends on the noise level. The D dimensions of the model may include any time-independent parameters, by promoting the parameters to state variables with time evolution given by $\dot{\mathbf{p}} = 0$.

The second ingredient is a time series of measurements, $\mathbf{Y} = \mathbf{y}_{1:M} = \{\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_M\}$. At each time step labeled by $n \in \{1, 2, \dots, M\}$ the measurement is an L -dimensional vector \mathbf{y}_n . Part of the model development consists of associating a known ‘observation’ function $\mathbf{h}(\mathbf{x}_n)$ of the model state variables \mathbf{x}_n with the observation \mathbf{y}_n . Usually one assumes that the observations and the observation function are related by additive noise added to the observation, so $\mathbf{y}_n \approx \mathbf{h}(\mathbf{x}_n) + \text{noise}$, but we first discuss a more general case. Typically $L < D$, so there are ‘hidden states’ which must be inferred using all the information available from observations and from the model. More keeping with a

physicist's view of measurements, we refer to the $D - L$ unmeasured states as unobserved state variables. We need to estimate them as well as the fixed parameters. We will assume later that the observation only depends on the state at the current time step.

We then consider the discrete time evolution of the model state by defining a path variable $\mathbf{X} = \mathbf{x}_{0:M} = \{\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_M\}$. This describes a trajectory through the D -dimensional state space. This formulation considers the entire path simultaneously, assuming all the observations have already been collected. In signal processing language this method would be called a 'smoother', as opposed to a 'filter' which only uses observations from the past.

We then use the Markov transition probabilities, $P(\mathbf{x}_n|\mathbf{x}_{n-1})$ specified by the model to find $P(\mathbf{x}_n|\mathbf{y}_{1:n-1})$ from $P(\mathbf{x}_{n-1}|\mathbf{y}_{1:n-1})$. We also use Bayes' rule [14] to introduce the observations into the formulation. The result is

$$\begin{aligned} P(\mathbf{x}_n|\mathbf{y}_{1:n}) &= \frac{P(\mathbf{y}_n|\mathbf{x}_n, \mathbf{y}_{1:n-1})}{P(\mathbf{y}_n|\mathbf{y}_{1:n-1})} \int d\mathbf{x}_{n-1} P(\mathbf{x}_n|\mathbf{x}_{n-1}) P(\mathbf{x}_{n-1}|\mathbf{y}_{1:n-1}), \\ &= \frac{P(\mathbf{y}_n, \mathbf{x}_n|\mathbf{y}_{1:n-1})}{P(\mathbf{x}_n|\mathbf{y}_{1:n-1}) P(\mathbf{y}_n|\mathbf{y}_{1:n-1})} \int d\mathbf{x}_{n-1} P(\mathbf{x}_n|\mathbf{x}_{n-1}) P(\mathbf{x}_{n-1}|\mathbf{y}_{1:n-1}) \\ &= e^{\text{CMI}(\mathbf{x}_n, \mathbf{y}_n|\mathbf{y}_{1:n-1})} \int d\mathbf{x}_{n-1} P(\mathbf{x}_n|\mathbf{x}_{n-1}) P(\mathbf{x}_{n-1}|\mathbf{y}_{1:n-1}), \end{aligned} \quad (1)$$

where we identify the factor in front of the integral as the exponential of the conditional mutual information between the state \mathbf{x}_n at time t_n and the measurement \mathbf{y}_n at that time.

This is the key equation that is used by all recursive Bayesian estimation methods. It is the starting point for particle filter methods [5] which use this equation to perform the forecast step to advance the probability distribution, as approximated by a weighted particle distribution, and the analysis step to incorporate measurements. Particle filtering methods process the measurements and update the model state sequentially. Here we take a different approach and consider the entire model evolution and the whole time series of observations simultaneously.

To write $P(\mathbf{x}_M|\mathbf{y}_{1:M})$ over a path in state space we first look at the initial time step by setting $n = 1$ into Eq. (1) to arrive at

$$P(\mathbf{x}_1|\mathbf{y}_{1:1}) = \frac{P(\mathbf{y}_1|\mathbf{x}_1)}{P(\mathbf{y}_1)} \int d\mathbf{x}_0 P(\mathbf{x}_1|\mathbf{x}_0) P(\mathbf{x}_0),$$

and we note that $\mathbf{y}_{1:0}$ means there are no measurements to condition the probabilities by. We then apply Eq. (1) iteratively $M - 1$ additional times to establish

$$P(\mathbf{x}_M|\mathbf{y}_{1:M}) = \int \prod_{n=1}^M d\mathbf{x}_{n-1} \frac{P(\mathbf{y}_n|\mathbf{x}_n, \mathbf{y}_{1:n-1})}{P(\mathbf{y}_n|\mathbf{y}_{1:n-1})} P(\mathbf{x}_n|\mathbf{x}_{n-1}) P(\mathbf{x}_0). \quad (2)$$

This is the marginal distribution of the state at the last time point \mathbf{x}_M .

This is also the integral representation of the solution to an underlying Fokker-Planck equation for the problem of a state \mathbf{x}_n satisfying noisy dynamical equations being informed by noisy measurements \mathbf{y}_n . When the noise is Gaussian the underlying Fokker-Planck equation is known as the Zakai equation [15].

We can use the definition of a marginal distribution

$$P(\mathbf{x}_M|\mathbf{y}_{1:M}) = \int d\mathbf{x}_0 \dots d\mathbf{x}_{M-1} P(\mathbf{x}_{0:M}|\mathbf{y}_{1:M}),$$

to express the conditional probability distribution as a function of the entire path $\mathbf{x}_{0:M}$ via

$$P(\mathbf{x}_{0:M}|\mathbf{y}_{1:M}) = P(\mathbf{X}|\mathbf{Y}) = \prod_{n=1}^M \left[\frac{P(\mathbf{y}_n|\mathbf{x}_n, \mathbf{y}_{1:n-1})}{P(\mathbf{y}_n|\mathbf{y}_{1:n-1})} \right] P(\mathbf{x}_n|\mathbf{x}_{n-1}) P(\mathbf{x}_0). \quad (3)$$

To calculate the marginal distribution at a particular time step $0 \leq n \leq M$, $P(\mathbf{x}_n|\mathbf{y}_{1:M})$, we integrate over the state variables at all the other time steps. Note that this formulation for $P(\mathbf{x}_n|\mathbf{y}_{1:M})$ includes the information gained from the whole time series of measurements $\mathbf{y}_{1:M}$ and not just the the past measurements $\mathbf{y}_{1:n}$.

We define the action, A , by taking the log of Eq. (3) and dropping additive terms independent of \mathbf{X} ,

$$-A(\mathbf{X}|\mathbf{Y}) = \sum_{n=1}^M \log P(\mathbf{y}_n|\mathbf{x}_n, \mathbf{y}_{1:n-1}) + \sum_{n=1}^M \log P(\mathbf{x}_n|\mathbf{x}_{n-1}) + \log P(\mathbf{x}_0). \quad (4)$$

The first term is where the information from the measurements is incorporated, the second term is where the dynamical model is incorporated, and the last term is the prior distribution of initial states and parameters, which takes into account any previous assumptions. Using this definition, we can write

$$P(\mathbf{X}|\mathbf{Y}) \propto e^{-A(\mathbf{X}|\mathbf{Y})}.$$

The constant of proportionality is independent of \mathbf{X} , so the expectation of any function $G(\mathbf{X})$ on the path \mathbf{X} , conditioned on the measurements \mathbf{Y} , is written as

$$E[G(\mathbf{X})|\mathbf{Y}] = \langle G(\mathbf{X}) \rangle = \frac{\int d\mathbf{X} G(\mathbf{X}) e^{-A(\mathbf{X}|\mathbf{Y})}}{\int d\mathbf{X} e^{-A(\mathbf{X}|\mathbf{Y})}}. \quad (5)$$

2.1. Assumptions leading to simplified form of action

We assume that the measurement at time step n depends in a known and deterministic way given by \mathbf{h} on the current state, \mathbf{x}_n , plus multivariate Gaussian noise. We also assume that \mathbf{y}_n is independent of any earlier measurement, and so

$$\begin{aligned} \mathbf{y}_n - \mathbf{h}(\mathbf{x}_n) &= \mathcal{N}(0, \mathbf{R}_\mathbf{o}^{-1}) \\ &\equiv \boldsymbol{\delta}_n, \end{aligned} \quad (6)$$

where $\mathcal{N}(0, \mathbf{R}_o^{-1})$ represents a random vector drawn from a multivariate Gaussian with zero mean and covariance matrix \mathbf{R}_o^{-1} . We these assumptions, the term inside the first sum of Eq. (4) becomes

$$P(\mathbf{y}_n | \mathbf{x}_n, \mathbf{y}_{1:n-1}) = P(\mathbf{y}_n | \mathbf{x}_n) \propto \exp \left[-\frac{1}{2} \boldsymbol{\delta}_n^T \cdot \mathbf{R}_o \cdot \boldsymbol{\delta}_n \right].$$

Next we make some assumptions about the dynamics. We define the model error term by discretizing the model differential equation, $\frac{d\mathbf{x}}{dt} = \mathbf{F}(\mathbf{x}(t))$, using the trapezoid rule with time step Δt :

$$\boldsymbol{\epsilon}_n \equiv \mathbf{x}_n - \mathbf{x}_{n-1} - \frac{\Delta t}{2} [\mathbf{F}(\mathbf{x}_n) + \mathbf{F}(\mathbf{x}_{n-1})]. \quad (7)$$

If the dynamics were deterministic, then $\boldsymbol{\epsilon}_n = 0$ (exact in the limit where $\Delta t \rightarrow 0$) for all n . We allow noise into the dynamics by relaxing this condition by replacing the zero with a stochastic term:

$$\boldsymbol{\epsilon}_n = \mathcal{N}(0, \mathbf{R}_d^{-1}).$$

We have allowed for some types of random error in the model by introducing a noise term, which is an additive Gaussian noise applied at every time step. The noise is parameterized by the covariance matrix $\mathbf{R}_d^{-1} = \Delta t \boldsymbol{\Gamma}$, where $\boldsymbol{\Gamma}$ is a diffusion matrix. We these assumptions, the term inside the second sum of Eq. (4) becomes

$$P(\mathbf{x}_n | \mathbf{x}_{n-1}) \propto \exp \left[-\frac{1}{2} \boldsymbol{\epsilon}_n^T \cdot \mathbf{R}_d \cdot \boldsymbol{\epsilon}_n \right].$$

Finally, for simplicity we assume that $P(\mathbf{x}_0)$ is uniform inside the boundaries and zero outside. This means the initial states and parameters can be restricted to fall inside a boundary, and that the last term in Eq. (4) can be dropped because it is a constant inside the boundary.

The action can be written as $A = A_o + A_d$, where

$$A_o = \frac{1}{2} \sum_{n=1}^M \boldsymbol{\delta}_n^T \cdot \mathbf{R}_o \cdot \boldsymbol{\delta}_n,$$

$$A_d = \frac{1}{2} \sum_{n=1}^M \boldsymbol{\epsilon}_n^T \cdot \mathbf{R}_d \cdot \boldsymbol{\epsilon}_n,$$

are the contributions from the observations and the dynamics respectively.

3. Monte Carlo Evaluation

Now that we have a formulation of $P(\mathbf{X}|\mathbf{Y})$, we would like to calculate quantities such as means and covariances, which can be used to make estimates and predictions. These quantities can be written as path integrals of the form

given in Eq. (5), where $G(\mathbf{X})$ is chosen to be some function of the path that is of interest.

The challenge is to evaluate these path integrals. One way to do this is to generate a series of paths $\{\mathbf{X}^{(1)}, \dots, \mathbf{X}^{(J)}\}$ that are distributed in path space according to $P(\mathbf{X}|\mathbf{Y}) \propto \exp[-A(\mathbf{X}|\mathbf{Y})]$, then we can use those paths to approximate the distribution with

$$P(\mathbf{X}|\mathbf{Y}) \approx \frac{1}{J} \sum_{j=1}^J \delta(\mathbf{X} - \mathbf{X}^{(j)}).$$

We can then calculate expectation values of any function of the path with

$$\begin{aligned} \langle G(\mathbf{X}) \rangle &= \int d\mathbf{X} G(\mathbf{X}) P(\mathbf{X}|\mathbf{Y}) \\ &\approx \frac{1}{J} \sum_{j=1}^J G(\mathbf{X}^{(j)}). \end{aligned} \tag{8}$$

The sample paths can be thought of as representing many possible time evolutions of the system state that could have produced the observed data when measured. The paths that are more likely to have produced the observed data will be generated more times than the paths which are less likely to have produced the observed data.

Now we need a method which will produce a series of paths that are distributed according to $\exp[-A(\mathbf{X})]$. There are several path integral Monte Carlo methods, such as Metropolis or Hybrid Monte Carlo, that do exactly this [16, 14, 17].

3.1. Metropolis Monte Carlo

One of the simplest and oldest approaches is the Metropolis Monte Carlo method [16]. This method works by generating a sequence of paths $\{\mathbf{X}^{(1)}, \dots, \mathbf{X}^{(J)}\}$ that we will call Monte Carlo paths by a random walk through path space. The method is an example of a Markov Chain Monte Carlo method, because the paths are generated in sequence and the next path is generated only from the current path in a stochastic way. The random walk is biased in a particular way as described below so that the sequence of paths that are generated come out distributed according to $\exp[-A(\mathbf{X})]$.

The method works by generating a new path $\mathbf{X}^{(n+1)}$ from the current path $\mathbf{X}^{(n)}$ with a two step procedure. First, a candidate path \mathbf{X}' is proposed by adding an unbiased random displacement to the current path $\mathbf{X} = \mathbf{X}^{(n)}$. The displacement may be to only one component or all the components, and may be drawn from any type of distribution, as long as it is unbiased; this assures that $\mathbf{X} \rightarrow \mathbf{X}'$ is as likely to occur as $\mathbf{X}' \rightarrow \mathbf{X}$.

The proposal distribution actually does not need to be unbiased if the acceptance probability is modified in the way shown in [18]. There are several methods which make a smarter choice of the proposal distribution, such as the

force bias method [19]. Such methods may converge with fewer iterations, but they have the added complexity of requiring derivatives of the action to be computed.

Next the proposed path is either accepted ($\mathbf{X}^{(n+1)} = \mathbf{X}'$) or rejected ($\mathbf{X}^{(n+1)} = \mathbf{X}$). The probability for acceptance is

$$P_{accept}(\mathbf{X}', \mathbf{X}) = \min(1, \exp[-\Delta A(\mathbf{X}', \mathbf{X})]), \quad (9)$$

where $\Delta A(\mathbf{X}', \mathbf{X}) = A(\mathbf{X}') - A(\mathbf{X})$ is the change in action which would be caused by changing \mathbf{X} to \mathbf{X}' . This says that if a proposed change will lower the action it should be accepted, and if the proposed change increases the action it should only be accepted with probability $\exp[-\Delta A(\mathbf{X}', \mathbf{X})]$. Note that only the change in action is required, so the full action never needs to be computed. This means we do not need to keep track of additive constants to the action, and can save a lot of computation time by only computing the terms in the action that will be changed by the update.

4. Parallel Implementation for GPUs

The Metropolis Monte Carlo method is simple and powerful, but it has the drawback of requiring very many path updates to get accurate statistics. Since so many path updates are required, the computation becomes very expensive in terms of computer time. One way to deal with this problem is to take advantage of parallel computing technology, using a Graphics Processing Unit (GPU). With GPU technology, and using Compute Unified Device Architecture (CUDA), it is possible to execute hundreds of threads of execution simultaneously. Typically each thread will perform the same operations, but on different pieces of the data. Of course since the paths are updated sequentially, the path update process cannot be run in parallel. However, the many computations needed on each iteration can be done in parallel by having different threads work on different time steps.

4.1. The general procedure

First the current path \mathbf{X} is set to an initial guess, and the observation time series \mathbf{Y} is loaded from a file. The path \mathbf{X} includes the state vector \mathbf{x}_n at every time step $n = 0, 1, \dots, M$, and the parameters. The states and parameters are treated differently, because the parameters are forced to be time-independent but the states may vary in time. The current path, the observation time series, the external drive signal (if present), and a running sum of moments of the path components are allocated in GPU memory and initialized with the appropriate data.

Then the path update loop begins. First the even n states are updated, and then the odd n states (see. Fig. 1 and Fig. 2). The reason for doing the state update in two steps is to uncouple the state vectors: to calculate the change in action due to perturbing \mathbf{x}_n , we need to know \mathbf{x}_{n-1} and \mathbf{x}_{n+1} , but none of the

1. For each odd n , a thread does the following:
 - (a) Proposed change: $\mathbf{x}'_n = \mathbf{x}_n + \Delta \mathbf{n} \cdot \mathbf{U}(-1, 1)$
 - (b) Calculate and store new RHS of model equations: $\mathbf{F}'_n = \mathbf{F}(\mathbf{x}'_n)$
 - (c) Calculate model error terms: $\epsilon_n, \epsilon'_n, \epsilon_{n+1}, \epsilon'_{n+1}$
 - (d) Calculate observation error terms: δ_n, δ'_n
 - (e) Change in dynamical part of action:

$$\Delta A_{d,n} = \frac{R_d}{2} [(\epsilon'_n)^2 - \epsilon_n^2 + (\epsilon'_{n+1})^2 - \epsilon_{n+1}^2]$$
 - (f) Change in observation part of action: $\Delta A_{o,n} = \frac{R_o}{2} [(\delta'_n)^2 - \delta_n^2]$
 - (g) Total change in action: $\Delta A_n = \Delta A_{d,n} + \Delta A_{o,n}$
 - (h) Acceptance probability: $P_{acc} = \min(1, e^{-\Delta A_n})$
 - (i) If $U(0, 1) < P_{acc}$ then accept the change: $\mathbf{x}'_n \rightarrow \mathbf{x}_n$ and $\mathbf{F}'_n \rightarrow \mathbf{F}_n$
2. For each even n , a thread executes the same steps shown above
3. Perturb parameter p_l . Launch $M + 1$ threads to compute the resulting change in action, and then either accept or reject the change with the usual Metropolis rule. Repeat for each parameter, $l = 1, 2, \dots, N_p$ in sequence.

Figure 1: Pseudo-code for parallel state and parameter update. For simplicity the matrices \mathbf{R}_o and \mathbf{R}_d are set to be R_o and R_d times the identity matrix, respectively. $U(a, b)$ is a random number between a and b drawn from a uniform distribution, and \mathbf{U} is a vector of D such components.

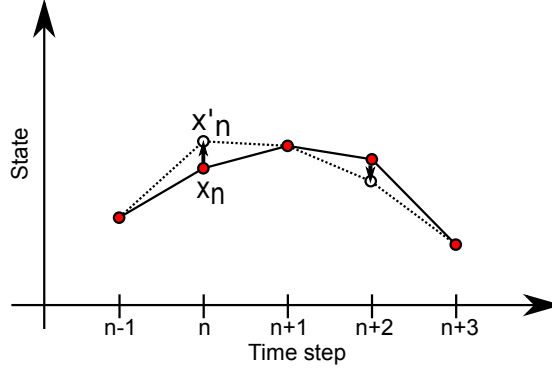


Figure 2: Schematic of the state update process. All even time steps are updated simultaneously by different threads. The solid circles show the current states (for illustration they are shown as scalars, but generally are vectors). The open circles show the proposed new states, and the dotted lines represent the dynamical terms in the action which would change. After all even time steps have a chance to update, the same procedure is done to all the odd time steps.

other state vectors. This way each even n , and then each odd n can be updated independently, in any order or simultaneously.

Once all the states have had a chance to be updated, then each parameter p_l is given a chance to change in sequence, $l = 1, 2, \dots, N_p$. One parameter, p_l , is perturbed, and then $M + 1$ threads are launched to calculate the change in action. Each thread is assigned to one n , and it calculates the new RHS of the model differential equation $\mathbf{F}'_n = \mathbf{F}(\mathbf{x}'_n)$. From this information the total change in action is calculated, and one decision about whether to accept the proposed change or not is made using the usual Metropolis rule. If the change is accepted, then $\mathbf{F}'_n \rightarrow \mathbf{F}_n$ for all n and $p'_l \rightarrow p_l$.

After all the states and parameters have had an opportunity to update, the current path can be used to update the path statistics. Typically this will be the mean and variance of each component of the path, but covariances or higher moments could be of interest too. Another possibility is to record bin counts of some components of interest to make a histogram. This step is skipped for the first N_{init} path updates, and after that only done every N_{skip} -th path update. The statistics collection happens on the GPU, also in parallel, and so the individual paths are not recorded. This avoids costly data transfers between the GPU and CPU.

4.2. More details

There are several more details which will now be discussed. The first N_{init} iterations are the initialization phase, during which no path statistics are recorded. This phase is necessary to remove the influence of the initial guess path. Two other things happen during this phase: simulated annealing, and automatic adjustment of the MC step sizes Δ_i .

The simulated annealing is done by putting a multiplier $\beta < 1$ in front of the dynamics term of the action, A_d , and gradually increasing β during the initialization phase, up to its final value of $\beta = 1$. This way the action will initially be dominated by the quadratic observation term, A_o , which is smooth and has a single well-defined minimum. Specifically, β is initially set to $\beta = \beta_0 < 1$ and multiplied by the constant factor f_β after each iteration, until $\beta = 1$. The constant multiplying factor is

$$f_\beta = (1/\beta_0)^{(1/N_{cool})}. \quad (10)$$

The β plays a role similar to inverse temperature T in a system with a Boltzmann distribution $P(\mathbf{x}) \propto \exp(-E(\mathbf{x})/kT)$. In our case the conditional path distribution is $P(\mathbf{X}|\mathbf{Y}) \propto \exp(-A_o(\mathbf{X}|\mathbf{Y})) \exp(-\beta A_d(\mathbf{X}))$. The two situations are not quite the same because in the second case there is a path-dependent and observation-dependent term, $A_o(\mathbf{X}|\mathbf{Y})$ which is not multiplied by β .

An automatic procedure for adjusting the the MC step sizes, Δ_i , is useful because there are typically many different step sizes which would be hard to tune by hand. It is important to tune the sizes because if they are too small, then the proposed path changes will be very small. They will be likely to be accepted, but very many iterations will be required for the path to change significantly.

On the other hand, if the step size is too large, most proposed changes will be rejected, and in this case also, many iterations will be required for the path to change significantly. In this case there is a different step size for each component of the path indexed by i . The delta adjustment rule used was

$$\Delta_i \leftarrow \Delta_i \left[1 + \alpha \left(\frac{N_{acc,i}}{N_{skip}} - f_{acc} \right) \right],$$

where $N_{acc,i}$ is the number of accepted changes for component i over the past N_{skip} number of iterations, f_{acc} is the target acceptance rate, and α is a constant that controls the adjustment rate. For a different approach which utilizes a proportional integral controller to tune step sizes see [20]. This rule is applied every N_{skip} -th iteration, and only during the initialization phase. It is important to note that adjusting the step sizes will bias the distribution, so samples generated during the adjustment phase should not be used when calculating statistics. It is run in parallel on the GPU, with one thread assigned to each path component. It was shown in [21, 22] that the optimal value for the acceptance rate is 0.23 in the limit of a infinite dimensional multivariate normal target distribution, so that is the target acceptance rate that we use here (see also [23]). It is important that each parameter and state variable are tuned independently, because the scales may be very different.

Another useful thing to do is to set boundaries on parameters. This can be done by adding a large penalty to the action when a proposed change would move the parameter outside of the boundary. This makes the proposed change be rejected. This is one way of incorporating a prior distribution of parameters, in the simple case where the distribution is uniform with the bounds and zero outside of the bounds.

The random number generation is done in parallel using the “Hybrid Tausworthe” algorithm described in [24]. Each thread is given its own initial seed, and so each thread will generate a different pseudo-random sequence. To test the quality of the random number generator we generated 512 different sequences in parallel of 10,000 random numbers each, and then computed the cross-correlations and autocorrelations. All correlations came out between -0.04 and 0.04, except the autocorrelations at zero time lag, which was one by choice of normalization.

4.3. GPU-specific details

There are several important limitations to keep in mind when designing CUDA code. The threads are grouped into blocks which run together. The amount of threads allowed in a block depends on how many resources each thread uses (often limited by shared memory usage or by number of registers used), but its maximum possible value is 1024 threads per block. The threads within a block can communicate with each other through shared memory, and can synchronize with each other. In contrast, threads in different blocks can have no interaction, and may be run in any order or simultaneously, so the code must be designed such that the order of block execution does not matter.

The GPUs have a large amount of global memory which is accessible to all threads, but has relatively high latency. The CPU can read or write to the global GPU memory, but this process should be kept to a minimum, because this process is relatively slow. Each block has a small amount of shared memory (49KB on the NVIDIA GTX 460 for example) which is much faster. The shared memory should be used where possible for temporary storage within the kernel call, but since its size is limited, there is often a trade off between using shared memory and being able to run more threads per block.

A common approach, which was used in the state update kernel, is to load the necessary pieces of information from global memory into shared memory at the beginning of the kernel. In this case the data was the \mathbf{F}_n values and the \mathbf{x}_n values for the relevant time steps. There is a overlap of one time step on the data which is read (but not of the data which is changed) between each block to make the neighboring values accessible. Since the blocks cannot synchronize with each other, it is necessary to do the even and odd n values in two separate kernel invocations. This ensures that all the even n finish before any odd n start.

The update process for the parameters is slightly more tricky, because in this case only one decision about whether to accept or reject is made, but it is based on the calculation of several blocks. This is handled by having each block calculate a change in action due to its assigned time steps. Each thread within the block calculates a change in action, and then participates in a parallel sum to get a total change in action for the whole block. Then after all the blocks finish another kernel is called which runs on a single thread and sums the block sums and decides whether to accept the change. If the change is accepted, the parameter is updated, and then the \mathbf{F}_n values are updated. Actually two copies of the \mathbf{F}_n 's are kept in global memory, one which was calculated from the current path and one which was calculated from the proposed path. There is a pointer to select which copy is current, and this pointer is flipped when the change is accepted, so that large memory transfers are avoided.

The observation time series and external drive signal (if present) are loaded into constant memory, which is a small (65KB on NVIDIA GTX 460) memory which can be quickly read by the GPU but not written to. For longer time series, this memory may be too small, in which case global memory is used instead. Also the data is organized in memory so that threads with contiguous indices will access contiguous regions of memory. This allows the memory transfers to be coalesced and happen more efficiently. All calculations were done using single-precision floating point arithmetic.

5. Results

5.1. Example problem: Hodgkin-Huxley

We now discuss the Hodgkin-Huxley neuronal model as an example problem. For other approaches to the similar problems see [25, 26]. The model, first developed in 1952 based on experiments on the squid giant axon [27], treats the

cell membrane as a capacitative layer with three types of conducting channels. The conductance of the sodium and potassium ion channels depends on the voltage across the membrane $V(t)$. The differential equation for voltage is:

$$\begin{aligned} \frac{dV}{dt} = & p_1 I_{stim}(t) \\ & + p_2 m^3(t) h(t) (p_3 - V(t)) \\ & + p_4 n^4(t) (p_5 - V(t)) \\ & + p_6 (p_7 - V(t)). \end{aligned} \quad (11)$$

The first term is an external stimulus current injected into the cell, the second term is the sodium ion current, the third term is potassium ion current, and the last term is a leak current. The voltage dependence of the conducting channels is modeled through gating probabilities $m(t), n(t), h(t)$, all between zero and one. The dynamics of the gating probabilities is specified by first-order kinetic equations with opening and closing rates which are functions of V . Equivalently, the equations can be expressed in terms of a steady state value $a_\infty(V)$ and a time constant $\tau_a(V)$, where a stands for m, n , or h :

$$\frac{da}{dt} = \frac{a_\infty(V) - a(t)}{\tau_a(V)}. \quad (12)$$

Phenomenological analytic expressions for these functions were discussed empirically by Hodgkin and Huxley. For this example we use different but qualitatively similar functions:

$$a_\infty(V) = \frac{1}{2} + \frac{1}{2} \tanh\left(\frac{V - V_a}{\Delta V_a}\right) \quad (13)$$

$$\tau_a(V) = \tau_{a0} + \tau_{a1} \left(1 - \tanh^2\left(\frac{V - V_a}{\Delta V_a}\right)\right). \quad (14)$$

Simulated data was created by integrating the Hodgkin-Huxley model, with a chaotic stimulus current generated from one of the variables of the Lorenz model [28]. Values for the parameters in the gating equation were chosen to qualitatively match what was found by Hodgkin and Huxley, and are given in Table 1. Simulated measurements of transmembrane voltage $V(t)$ were generated from the model output with a sampling frequency of 25 kHz. This simulated data along with the chaotic stimulus current is shown in Fig. 3.

The model has a total of four state variables, V, n, m , and h , but only the voltage is observed. This corresponds to the typical situation in real experiments. In a current clamp experiment on a single isolated neuron, a known current is injected into the cell while the transmembrane voltage response is recorded. In this version of the model, seven numbers p_1, p_2, \dots, p_7 , were treated as unknown parameters, and all other constants were fixed at the same values for data generation and for the analysis. One can determine all of the constants

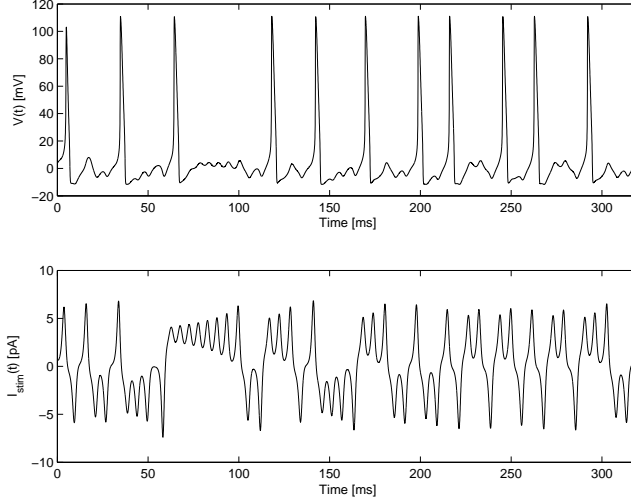


Figure 3: The simulated voltage measurement and the chaotic stimulus current. These two time series plus the Hodgkin- Huxley model (with seven unknown parameters) are inputs into the PIMC procedure. The $V(t)$ data is 8000 discrete points (although it is shown as a solid line) sampled at 25 kHz.

in $a_\infty(V)$ and $\tau_s a(V)$ with the same methods. Our focus is on the computational procedures here. In the present case the observation function $\mathbf{h}(\mathbf{x}_n)$ simply returns the the voltage component of \mathbf{x}_n , but in general it could be any function of \mathbf{x}_n .

In the example calculation, one million path update iterations were performed, with $N_{skip} = 400$. The first half million were excluded from the statistics ($N_{init} = 500,000$). Simulated annealing was done over the first $N_{cool} = 100,000$ iterations, with $\beta_0 = 0.01$, and β was incremented by multiplying by f_β (Eq. 10) after each iteration. The target acceptance rate was $f_{acc} = 0.23$ and the step size adjustment rate was $\alpha = 0.02$. The initial settings for Monte Carlo step sizes were $\Delta_i = 2 \times 10^{-3}$ for voltage and $\Delta_i = 10^{-3}$ for the n, m , and h state variables.

The number of data points used was $M = 8000$, or 320 ms of data. The matrix \mathbf{R}_d was set to be $[\mathbf{R}_d]_{V,V} = 100$, $[\mathbf{R}_d]_{a,a} = 10^6$ ($a \in \{n, m, h\}$), and all other components zero. Since the observation is one-dimensional \mathbf{R}_o is a scalar, and was set to 100. Figure 4 shows the evolution of one parameter, p_1 , to give an idea of the equilibration process. The mean and standard deviation of each component of the path, that is each state at each time step and each parameter, was calculated. The mean and the standard deviation of the unobserved states shown in Fig. 5. The estimated parameters values are given in Table 2. This calculation took 683 seconds to run for one million iterations on an NVIDIA GTX 460 GPU which has 224 cores.

For timing purposes, the same calculation was done over a range of time series lengths from $M = 100$ up to $M = 40,000$, but with 1/10th as many

Table 1: Constants used in the kinetic equations.

Name	V_a [mV]	ΔV_a [mV]	τ_{a0} [ms]	τ_{a1} [ms]
n , K Activation	10	30	1.0	5.0
m , Na Activation	25	15	0.1	0.4
h , Na Inactivation	5	-15	1.0	7.0

Table 2: Estimated parameters.

Name	Mean	St. Dev.	Actual	Units
p_1	1.02	0.04	1	1/pF
p_2	127	2	120	1/ms
p_3	115	0.1	115	mV
p_4	20.3	0.3	20	1/ms
p_5	-12.0	0.08	-12	mV
p_6	0.33	0.03	0.3	1/ms
p_7	9.3	1.0	10.6	mV

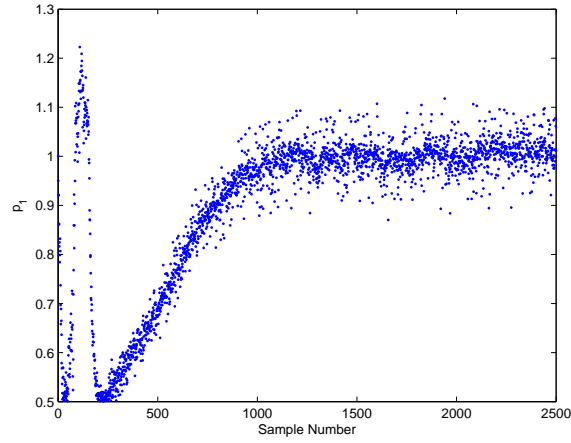


Figure 4: The evolution of one of the parameters as the MC path update procedure runs. The first 1250 samples were discarded. Simulated annealing was applied during the first 250 iterations. Eventually the parameter settles down and fluctuates around a mean of 1.02.

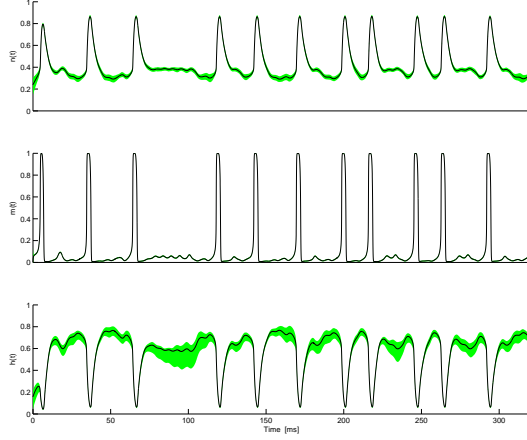


Figure 5: The output of the PIMC process. These are the three unobserved states. The black line is the true value (which is known because the data was simulated). The solid region is centered on the mean and extends to plus or minus 4 standard deviations from the mean.

iterations, and the execution time, T_{GPU} , was recorded. The same calculation was done on two different NVIDIA GPUs, the GTX 460 with 224 cores and the GTX 470 with 448 cores. In all cases, the number of threads per block was set to 100. For comparison a similar, but not exactly the same, calculation was done on a single core of an Intel Core i3 CPU. A linear time scaling of $T_{CPU} = M \times (1.24 \text{ s})$ was fit from several trials on the CPU. In Fig. 6 the parallel speedup factor, T_{CPU}/T_{GPU} , is displayed.

6. Discussion

Running the PIMC in parallel decreased the computation time by a factor of up to 200 or 300, depending on which GPU was used. As the time series becomes longer, more of the GPU cores are simultaneously utilized. The specifics of how the speedup factor varies as a function of time series length depends on the particular GPU, and on how CUDA schedules the threads to utilize the available resources.

One way to assess the quality of the model is to use the model to integrate the final state at time $t = T$ forward in time and compare to additional observations. One possibility is to use the mean final state $\langle x_T \rangle$ and mean value of the parameters, $\langle p_l \rangle$, to generate a single trajectory, which can then be compared to the observed time series. However, this may not be adequate when the dynamics is noisy, or when the model is chaotic. Another approach is to generate an ensemble forecast, with the appropriate amount of noise added into the model integration. Many noisy model integrations should be done, by using many different final states and parameter values as generated by the Monte Carlo

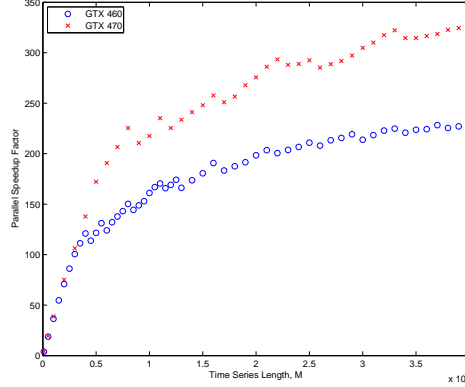


Figure 6: The parallel speedup factor as a function of time series length M , using two different NVIDIA GPUs. The parallel speedup factor is defined as T_{CPU}/T_{GPU} . The performance of the two GPUs is about the same for small M , where they are underutilized, but as M increases, the GTX 470 performs better because it has more cores.

procedure, and many different noise sequences. Then the ensemble forecast can be compared to the single observed trajectory in a statistical way.

Another way to assess the quality of the model is to look at the model errors, ϵ_n , and the observation errors, δ_n . These will have specific values for each Monte Carlo path generated, since they are functions of the path, \mathbf{X} . If the model is correct, then the statistics of these quantities should be consistent with our assumptions about the noise. In the example problem the assumptions are that both noises are Gaussian and white. Therefore the means should be zero, the variances should come out as specified by the \mathbf{R}_o and \mathbf{R}_d matrices, and there should be no correlation in time or among the different components. In the example problem presented here, there is a source of model error which comes from the discretization of the model differential equations. This appears as deviations of $\langle \epsilon_n \rangle$ from zero at the times when $V(t)$ spikes.

The method presented in this paper is quite general, and can be applied to any dynamical system. All that is needed is a time series of observations, a Markov model of the dynamics, and some assumptions about the noise in the dynamics and the noise in the observations. By utilizing GPU technology, it becomes practical to run the method on realistic problems using desktop computers.

Acknowledgments

We thank Marius Buibas and Reza Farsian for useful discussions about parallelizing the algorithm. Support from the US Department of Energy (Grant de-sc0002349) and the National Science Foundation (Grants IOS-0905076 and

PHY-0961153) are gratefully acknowledged. Partial support from the NSF sponsored Center for Theoretical Biological Physics is also appreciated.

References

- [1] H. Cox, On the estimation of state variables and parameters for noisy dynamic systems, *IEEE transactions on automatic control* (1964) 5–12.
- [2] B. Friedland, I. Bernstein, Estimation of the state of a non-linear process in the presence of nongaussian noise and disturbances, *Journal of the Franklin Institute* 281 (6).
- [3] R. Kalman, A new approach to linear filter and prediction problems, *J. Basic Eng.* 82 (1960) 35–45.
- [4] P. J. V. Leeuwen, Particle filtering in geophysical systems, *Monthly Weather Review* 137 (2009) 4089–4114.
- [5] S. Arulampalam, S. Maskell, N. Gordon, T. Clapp, A tutorial on particle filters for on-line non-linear/non-Gaussian Bayesian tracking, *IEEE Transactions on Signal Processing* 50 (2001) 174–188.
- [6] A. Doucet, S. Godsill, C. Andrieu, On sequential Monte Carlo sampling methods for Bayesian filtering, *Statistics and Computing* 10 (2000) 197–208.
- [7] F. J. Alexander, G. L. Eyink, J. M. Restrepo, Accelerated Monte Carlo for optimal estimation of time series, *Journal of Statistical Physics* 119 (5).
- [8] J. M. Restrepo, A path integral method for data assimilation, *Physica D: Nonlinear Phenomena* 237 (1) (2008) 14 – 27. doi:DOI:10.1016/j.physd.2007.07.020.
- [9] A. Apte, M. Hairer, A. Stuart, J. Voss, Sampling the posterior: An approach to non-Gaussian data assimilation, *Physica D: Nonlinear Phenomena* 230 (1-2) (2007) 50 – 64. doi:DOI:10.1016/j.physd.2006.06.009.
- [10] H. D. Abarbanel, Effective actions for statistical data assimilation, *Physics Letters A* 373 (44) (2009) 4044 – 4048. doi:DOI:10.1016/j.physleta.2009.08.072.
- [11] T. Preis, P. Virnau, W. Paul, J. J. Schneider, GPU accelerated Monte Carlo simulation of the 2d and 3d Ising model, *Journal of Computational Physics* 228 (12) (2009) 4468 – 4477.
- [12] D. T. Pham, Stochastic methods for sequential data assimilation in strongly nonlinear systems, *Mon. Weath. Rev.* 129 (2001) 1194–1207.
- [13] J. C. Quinn, H. D. Abarbanel, State and parameter estimation using Monte Carlo evaluation of path integrals, *Quarterly Journal of the Royal Meteorological Society* 136 (652) (2010) 1855–1867.

- [14] D. J. MacKay, Information Theory, Inference, and Learning Algorithms, Cambridge University Press, 2003.
- [15] M. Zakai, On the optimal filtering of diffusion processes, Probability Theory and Related Fields 11 (1969) 230–243.
- [16] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, E. Teller, Equations of state calculations by fast computing machines, J. Chem. Phys 21 (1953).
- [17] R. M. Neal, Probabilistic inference using Markov chain Monte Carlo methods, Technical Report CRG-TR-93-1.
- [18] W. K. Hastings, Monte Carlo sampling methods using Markov chains and their applications, Biometrika 57 (1) (1970) 97–109.
- [19] C. Pangali, M. Rao, B. Berne, On a novel monte carlo scheme for simulating water and aqueous solutions, Chemical Physics Letters 55 (3) (1978) 413 – 417. doi:DOI:10.1016/0009-2614(78)84003-2.
- [20] J. R. Banfelder, J. A. Speidel, M. Mezei, Automatic determination of step-size parameters in monte carlo simulation tested on a bromodomain-binding octapeptide, Algorithms 2 (1) (2009) 215–226. doi:10.3390/a2010215.
- [21] G. Roberts, J. Rosenthal, Optimal scaling for various Metropolis-Hastings algorithms, Statistical Science 16 (4) (2001) 351–367.
- [22] A. Gelman, G. Roberts, W. Gilks, Efficient Metropolis jumping rules, Bayesian Statistics (1996) 599–607.
- [23] R. H. Kincaid, H. A. Scheraga, Acceleration of convergence in monte carlo simulations of aqueous solutions using the metropolis algorithm. hydrophobic hydration of methane, Journal of Computational Chemistry 3 (4) (1982) 525–547. doi:10.1002/jcc.540030410.
- [24] H. Nguyen, GPU Gems 3, NVIDIA Corporation, 2008.
- [25] Q. J. M. Huys, M. B. Ahrens, L. Paninski, Efficient Estimation of Detailed Single-Neuron Models, J Neurophysiol 96 (2) (2006) 872–890. doi:10.1152/jn.00079.2006.
- [26] M. Rodriguez-Fernandez, J. A. Egea, J. R. Banga, Novel metaheuristic for parameter estimation in nonlinear dynamic biological systems, BMC Bioinformatics 7 (2006) 483–501. doi:10.1186/1471-2105-7-483.
- [27] A. Hodgkin, A. Huxley, A quantitative description of membrane current and its application to conduction and excitation in nerve, J. Physiology (1952) 500–544.
- [28] E. N. Lorenz, Deterministic nonperiodic flow, Journal of the Atmospheric Sciences 20 (1963) 130–141.